Ph.D. Proposal:
Hierarchical Loadable Schedulers

John Regehr

April 27, 1999

1 Introduction

The processors in workstations, personal computers, and servers are becoming increasingly powerful, enabling them to run new kinds of applications, and to simultaneously run combinations of applications that were previously infeasible. However, fast hardware is not enough—the operating system must effectively manage system resources such as processor time, memory, and I/O bandwidth. The proposed work focuses on management of processor time, the effectiveness of which is an important factor in overall system performance [14].

Each processor in a system can only perform one task at a time; the *scheduler* in the operating system decides which task that is. The schedulers in general-purpose operating systems are designed to provide fast response time for interactive applications and high throughput for non-interactive ones. Unfortunately, these schedulers have very little knowledge of applications' actual CPU scheduling needs, causing them to poorly schedule several classes of applications. Even multi-gigahertz processors will be of little use if the operating system does not run the right applications at the right times.

Many domain-specific scheduling algorithms have been developed by the operating systems community. For example, real-time schedulers and accompanying analysis enable computations to complete before deadlines under certain conditions. Gang schedulers and implicit coschedulers permit efficient execution of parallel programs when the progress of a task is highly dependent on the progress of other tasks. Proportional share schedulers can prevent groups of tasks from interfering with other groups by limiting aggregate CPU usage of groups. Because the schedulers in general-purpose operating systems do not implement these specialized algorithms, users cannot reap the associated benefits—predictability, parallel speedup, and load isolation, in the examples above.

It is difficult to select, in advance, the right scheduler for an operating system because the choice of scheduling algorithm depends on the mix of applications. In fact, a premise of this work is that it *should not* be selected in advance. Rather, we enable code implementing arbitrary scheduling policies to be loaded into the kernel at run-time. Processor time is then allocated by the set of loaded schedulers in cooperation with applications, and a CPU resource manager which manages interactions between schedulers and enforces user-specified policies about processor allocation.

In this document, *task* and *application* are used interchangeably to describe the set of computations associated with some user application. A *thread* is a schedulable entity—a task is typically implemented by a group of cooperating threads.

The *hierarchical loadable scheduler architecture* has two parts: the *loadable scheduler infrastructure*, which is implemented in the operating system and provides the framework for loadable scheduling, and loadable schedulers, which implement the *scheduler interface*. Loadable schedulers may schedule threads or other schedulers; they are arranged in a hierarchy—processor time starts at a root scheduler and moves to the leaves, which are threads. Figure 1 shows a high-level view of the system.

2 Goals of the Proposed Work

The first goal of my work is to reduce the cost of adding application-specific scheduling policies to general purpose operating systems. The cost of adding a new scheduling policy to an operating sys-

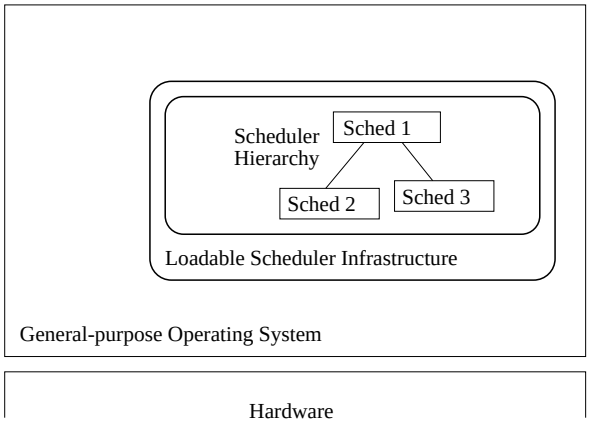

Figure 1: The Hierarchical Loadable Scheduler Architecture.

tem without loadable schedulers includes the cost of obtaining and understanding the source code for the OS, and the cost of integrating the new scheduler with existing scheduler code. The resulting monolithic scheduler is likely to be complex and difficult to maintain, as well as being inflexible—its policies cannot easily be replaced or composed in some other arrangement. The second goal is to allocate processor time to applications that the user or users deem important, using a high-level resource management approach.

My thesis statement, then, is:

> Hierarchical Loadable Scheduling defines an architecture for dynamic composition of scheduling policies in the kernel of a general-purpose operating system. The architecture enables the use of scheduling policies that closely match application requirements, while avoiding the cost of developing a complex, monolithic scheduler that implements all of the desired functionality, and the inflexibility inherent in that approach. Low-level scheduling decisions are made by the loaded schedulers, and high-level processor allocation decisions are made by a resource manager whose goal it is to maximize the value of the system as perceived by the user.

The challenge is to design the scheduler interface in such a way that the functionality of independently authored loadable schedulers can be composed, with predictable results. The system also needs to be efficient at run-time, and to effectively manage processor time in accordance with user policies.

Schedulers implementing the scheduler interface are to be source-compatible between operating systems that implement the loadable scheduler infrastructure. This is a desirable property because schedulers can be complex pieces of code and, if possible, should be written once and used many times. The Uniform Driver Interface [16] has similar goals—portability of device drivers between UNIX variants. However, it specifies compatibility at the binary level.

3 Motivation

Two examples, one from industry and one from academia, motivate the need for loadable scheduling policies. Intel's *Soft Migration* [4] initiative is an attempt to move functionality traditionally found on peripheral hardware onto the main CPU. For example, *soft modems* perform all signal processing in software—they are cheaper than regular modems, which require dedicated signal-processing hardware. Similarly, *SoftOSS* from 4Front Technologies [18] is a software implementation of a wavetable mixer. Both products are CPU-intensive, and both require timely execution—missed deadlines can result in modem resets and sound glitches.

Because they cannot count on predictable scheduling by the time-sharing schedulers in general purpose operating systems, both products run their time-dependent code in kernel drivers, completely bypassing the scheduler. This approach is undesirable for several reasons, but it will work as long as only one such driver is present in the system—when there are two or more, then conflicts between them can easily result in missed deadlines even when the system is underloaded. A system with a real-time scheduler would have the capability to either recognize that together, SoftOSS and a soft modem over-commit the processor resource, or to schedule both pieces of software in such a way that deadlines are never missed.

Banga et al. have added the *resource container* [2] abstraction to Digital UNIX, which enables the resources of a group of processes to be managed as a unit. This is useful, for example, on a machine

acting as a web server for two separate domains—the web server threads, CGI processes, etc. for each domain can be prevented from degrading the performance of other domains. Implementing resource containers required modifications to the CPU scheduler; the authors implemented, in effect, a fixed, two-level scheduling hierarchy.

In the first example, the problem was solved badly by approximating a real-time scheduler using device drivers. In the second, the authors had to modify a complex piece of code, the Digital UNIX scheduler, to achieve their goals. Both problems could have been solved more effectively in a system supporting loadable scheduling policies.

3.1 Related Work

Most modern operating systems permit code to be dynamically loaded into the kernel. However, it is usually the case that only device drivers may be loaded—the interfaces available to loadable kernel modules are quite limited.

Vassal [11] is an immediate intellectual predecessor to my work—it is a modified version of Windows NT that permits loadable kernel modules to make scheduling decisions; its decisions take priority over the NT scheduler, which runs when Vassal makes no decision. The static relationship between the schedulers, the limited kernel interface, and the ability to load only one scheduler at a time are limitations of the prototype Vassal implementation that will be removed in my work.

SPIN [3] allows users to create kernel support for their own thread packages by downloading code into the kernel. The thread scheduler is scheduled by a global scheduler, which can be replaced but not extended. Exokernel [6] gives applications direct control over context switching and scheduling. However, the global scheduler is again provided by the system, and is not extensible.

UNIX System V release 4 supports several scheduling policies called *scheduling classes* [8, pp. 160–191]. However, scheduling decisions are still mapped onto priorities, and the highest priority thread is selected for execution. Nieh et al. [14] have shown that scheduling classes are insufficient to effectively schedule multimedia applications.

CPU inheritance scheduling [7] is closely related to my work. It runs user-specified schedulers in separate, unprivileged protection domains; they communicate through an IPC-based interface. The basic primitive is CPU donation—scheduler threads "hand off" the CPU to other schedulers, and finally to a program thread. This allows flexible composition of scheduling policies, although there is no accompanying resource management framework to regulate allocation of CPU time.

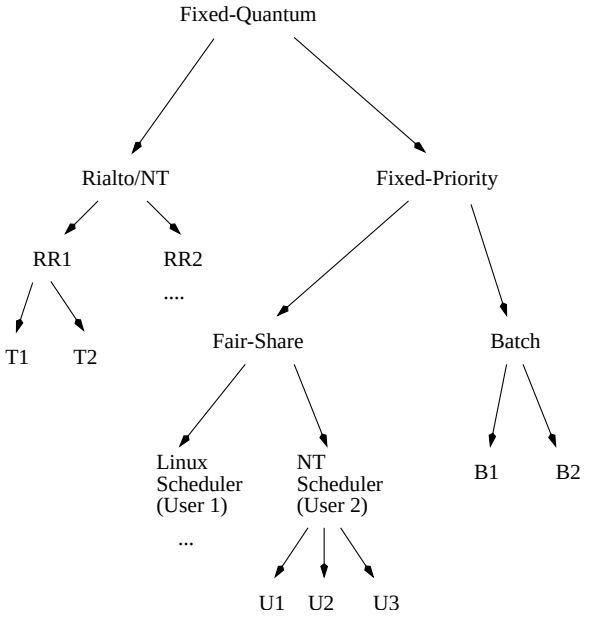

Figure 2: Example scheduling hierarchy.

Stankovic et al. have developed a hierarchical algorithm that uses a multimedia server, scheduled using a hard real-time scheduler, to run soft real-time applications [13, 17].

Goyal and Guo use start-time fair queuing to partition CPU time hierarchically, with other scheduling algorithms present at the leaf nodes of the scheduling tree [9]. Deng et al. describe a similar scheme, but using an EDF scheduler at the root of the scheduling tree [5].

The resource management aspects of my work are similar to those proposed by Jones for the Rialto operating system [10] and Oparah for Nemesis [15], although the tight integration of processor management and scheduling policies is new.

3.2 An Example

Figure 2 shows an example of a scheduling hierarchy, to which I will refer throughout this document. The fixed-quantum scheduler is a simple

reservation-based real-time scheduler that gives, for example, 5ms out of every 20ms of CPU time to the Rialto/NT scheduler, and the rest to the fixed-priority scheduler. Rialto/NT is a complex real-time scheduler that uses sub-schedulers (running a modified round-robin algorithm) to schedule *activities*—groups of threads that cooperate to perform a task.

The fixed-priority scheduler always runs the fair-share scheduler if possible, and runs the batch scheduler otherwise. The fair-share scheduler provides load-isolation between users, who can run the timesharing scheduler of their choice. User 1 is running a loadable version of the Linux scheduler, and user 2 runs a loadable version of the NT scheduler, which schedules threads U1, U2, and U3.

The batch scheduler gives long time-slices to the non-interactive jobs B1 and B2 in order to reduce context switch overhead. Note that in this example, when the real-time scheduler has anything to schedule, the "long time-slices" will be at most 15ms. In other words, the batch scheduler is only expected to increase efficiency when it has little competition from other schedulers.

This example illustrates three improvements to the scheduling capabilities of a general purpose operating system. First, it enables us to schedule real-time activities using the Rialto/NT scheduler. Second, it isolates the CPU usage of two users from each other. Third, it enables efficient execution of background tasks when the processor is otherwise idle. Although a single scheduler providing all of this functionality could be implemented, it would not only be difficult to develop and maintain, but it would be inflexible, and therefore difficult to compose into a new hierarchy that reflects changing needs of applications.

4 Design of the Loadable Scheduler Architecture

4.1 Assumptions

My work will focus on mainstream consumer multiprocessor operating systems such as Windows NT, Linux, Solaris, and FreeBSD. Which OSs I actually decide to work with will depend on source code availability and whether the structure of the OS internals make it easy to implement loadable schedulers.

I will use the loadable kernel module (LKM) functionality found in most modern operating systems. An LKM is a collection of functions and data that is loaded into the kernel address space at run time. Generally, a subset of the internal kernel APIs are available to LKMs; part of the proposed research is to figure out what additional functionality needs to be exported to LKMs to allow them to be schedulers.

Schedulers will be loaded into the kernel to avoid inefficiency caused by protection boundary crossings. In principle, there is no reason why the scheduler interface could not be used for user-space schedulers as well as in-kernel schedulers. Supporting this would require a second, IPC-based implementation of the scheduler interface, similar to the one developed by Ford and Susarla [7]. User-level schedulers would be easier to debug than in-kernel schedulers because more sophisticated tools are available in user-space, but context switch overhead is expected to be large.

Loaded schedulers will be managed separately from instances of schedulers in the hierarchy. Loaded schedulers are static blocks of kernel code, and instances contain the dynamic state associated with a scheduler: the position in the hierarchy, the set of threads being scheduled, guarantees made to those threads, and policy information. These are discussed in detail in following sections.

Every thread in the system belongs to, and is scheduled by, at least one scheduler; a thread might belong to more than one scheduler if, for example, it needs real-time scheduling but can also opportunistically use CPU time given to it by a timesharing scheduler. In the remainder of the document, I will say that a thread belongs to a scheduler if and only if the scheduler gives a guarantee to the thread.

4.2 Scheduler Interface Requirements

To support schedulers that implement the loadable scheduler interface, the scheduler infrastructure must:

- maintain the scheduler hierarchy—load and unload schedulers, add instances of schedulers into the hierarchy and remove them
- traverse the scheduling hierarchy to make a scheduling decision

- notify schedulers of events of interest—blocking and unblocking threads, available and preempted processors, etc.
- manage processor time according to user-specified policies
- support negotiation of guarantees between applications and schedulers
- provide services to schedulers, such as messaging, synchronization, inter-processor communication, memory allocation, timers, and accounting data
- provide services to threads—requesting a guarantee from a scheduler, negotiating a new guarantee, sending a message to a scheduler, etc.

4.3 Scheduler Interface Design

The design space for the loadable scheduler interface is similar to the design space for schedulers in user-level thread packages. In both situations, there is a tradeoff to make between a simple, efficient design which possibly does not notify the scheduler of some events of interest, and a less efficient, complex design which gives the scheduler more information. I want to make available all events of interest, but permit schedulers to ignore some of them. This is discussed in more detail in the next section.

Messages from threads to schedulers will probably be implemented by adding a system call to the OS; messages between schedulers are lightweight, and will be implemented as function calls or events placed in shared-memory queues. Messages from schedulers to threads will use the native asynchronous notification primitive provided by the OS—signals in UNIX and APCs in NT.

4.4 Flow of Control

Every scheduler provides a *query routine* that the scheduler infrastructure calls while making a scheduling decision. The query routine is called with the number of the processor being scheduled as an argument; the scheduler returns either a reference to a thread or another scheduler, or reports that it has nothing to run. To select a thread to run, the system queries schedulers starting at the root of the hierarchy. Once a scheduler returns a reference to a thread, the dispatcher restores the context of that thread and jumps to it.

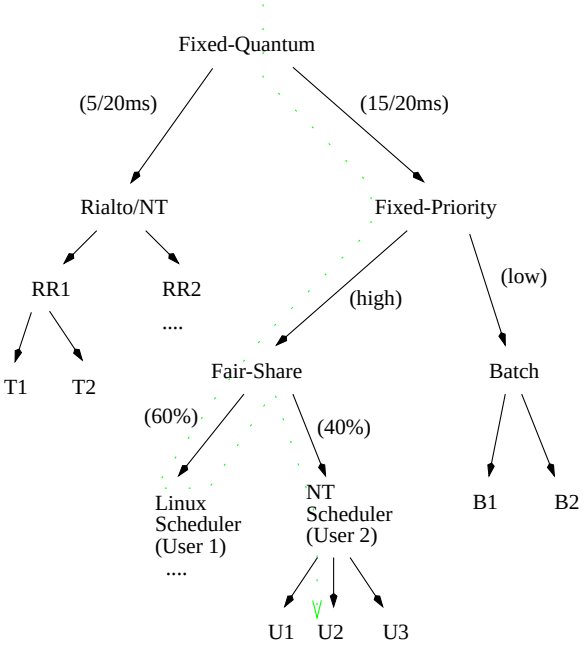

Figure 3: A path through the hierarchy.

Figure 3 illustrates this process, assuming a single processor for convenience. The fixed-quantum scheduler runs the fixed-priority scheduler, which runs the fair-share scheduler whenever possible. User 1 has no runnable threads, so the fair-share scheduler is queried again; it selects user 2's scheduler, and thread U2 is selected and run.

A naive implementation of this procedure will be inefficient, especially for large hierarchies. Fortunately, there are a number of optimizations that will make thread selection more efficient. Often, it is unnecessary to run the full selection process since the infrastructure can reuse the results of earlier computations when no events have happened that make them invalid. For example, if thread U2 blocks shortly after it starts running, and nothing has happened that affects the state of other schedulers, then the system can just notify the NT scheduler of the blocked thread and then query it for another thread to run. It should be possible to exploit partial results within schedulers as well as between them, to optimize for the case where a scheduler is called repeatedly as schedulers below it report that they have nothing to run.

To continue the example, assume that U3 begins running after U2 blocks. A short time later, a

timer interrupt wakes up the fixed-quantum scheduler, which now decides to run the Rialto/NT scheduler, which runs RR1, which runs T1. Before running T1, the system calls a function in the NT scheduler to notify the scheduler that it was preempted. Although a scheduler implementing scheduler activations [1] needs to know about preemptions in order to send an inter-processor interrupt, the NT scheduler does not, and will ignore the notification. The system can optimize away these notifications, and others, if the scheduler informs the infrastructure that it does not need them. Ford and Susarla describe similar optimizations for CPU inheritance scheduling [7].

If part of the scheduling hierarchy can be fixed at compile-time, then further optimizations are possible. In this case, the internal interfaces are unnecessary and can be optimized out of the code path. A special case of this could be realized by rolling certain simple and useful schedulers (e.g., the fixed-quantum and fixed-priority schedulers) into the scheduling infrastructure. They would still be conceptually loadable, but the fast internal implementation would be substituted for the loadable one.

4.5 Scheduler Guarantees

Many schedulers have the capability to make guarantees to threads that they schedule. These guarantees can be strong, in the case of real-time schedulers, or very weak for timesharing schedulers. For example, the Linux and NT schedulers simply guarantee that no runnable thread will be starved of CPU time (and even this does not apply if fixed-priority "real-time" threads are present).

The loadable scheduler infrastructure will provide explicit support for guarantees. Schedulers obtain guarantees from schedulers above them in the hierarchy, and make guarantees to threads (or other schedulers) that they schedule. For convenience, I call an agreement for a scheduler to schedule a thread a guarantee even when, as above, no non-trivial guarantee is present.

Schedulers can make guarantees no stronger than the ones they themselves receive. Therefore, real-time schedulers, which need to make strong guarantees, will be above non-real-time schedulers in the hierarchy.

Continuing with the example, the fixed-quantum scheduler is at the root of the hierarchy, and therefore always has control of what is being scheduled. It gives a guarantee of "5ms/20ms (hard)" to Rialto/NT, meaning that the same block of 5ms out of every 20ms will be available to it. A gang scheduler, for example, would require a slightly stronger guarantee that says *which* 5ms out of the 20ms it will receive, in order to coordinate with gang schedulers on other processors or machines. The Rialto/NT scheduler makes guarantees of the form "3ms/40ms (continuous)" to the Rialto sub-schedulers directly beneath it in the hierarchy—these schedulers represent user tasks. A task with this guarantee will run for 3ms out of any 40ms interval in time. The sub-schedulers in turn make guarantees of the form "2s by 17:00" to threads in the activity—a thread with this guarantee will receive 2 seconds of CPU time before 5pm. The exact format of guarantees is not important, as long as they are comprehensible by the system and the appropriate schedulers.

The fixed-priority scheduler gets a guarantee of "15ms/20ms (hard)". It passes this guarantee unchanged to the fair-share scheduler. The batch scheduler receives a null guarantee; threads it schedules cannot assume any minimum rate of progress.

The fair-share scheduler receives at least 15ms out of every 20ms, or $\frac{3}{4}$ of the CPU time. It gives 60% of this to user 1, and 40% to user 2. User 1 runs the Linux scheduler which makes no particular guarantees to its threads, but she is guaranteed that her threads will receive no less than 45% ($\frac{3}{4}$ of 60%) of the CPU over some time period determined by the fair-share scheduler. Similarly, user 2 runs the NT scheduler which makes no particular guarantees, but together his threads receive no less than 30% ($\frac{3}{4}$ of 40%) of the CPU.

When a scheduler is loaded into the hierarchy, the scheduling infrastructure will verify that the scheduler above it is capable of providing the kinds of guarantees that it needs. For example, the NT scheduler is never allowed to schedule a real-time scheduler.

To support legacy applications that are unaware of scheduler guarantees, new threads must be scheduled by a timesharing scheduler, which, in turn, must be given enough of a guarantee to permit these

applications to run with acceptable performance.

4.6 The Resource Manager

Engler et al. [6] observe that "mechanism *is* policy, albeit with one less layer of indirection." Although loadable schedulers enable the use of application specific scheduling policies, as far as users are concerned, the resulting low-level allocation of processor time is just a mechanism. The *resource manager* (RM) is the part of the scheduler infrastructure that is responsible for enforcing high-level policies about the management of processor time.

Applications send requests for guarantees to the RM, which selects an appropriate scheduler to provide the guarantee (loading a new scheduler if necessary). The purpose of the resource manager is to maximize the perceived value of the currently running set of applications to the user or users, within the constraints of user policies.

Call the set of running applications and their scheduler guarantees the *system configuration*. When an application requests a guarantee, it is asking the RM to switch to a new system configuration. If the new configuration would overload the system (by requesting 105% of the processor, for example), then the RM selects a different configuration which is feasible. The new configuration may be the same as the old one, in which case the new application's request for a guarantee is rejected, or it may involve revoking or reducing a guarantee made to a less important application. *Importance* is a high-level analogue of priority.

The user must determine the relative importance of applications, although I expect that default values will be used in the majority of cases. For example, a process controlling a CD writer would usually be considered to be very important—the cost of a missed deadline is a ruined disc. Some applications will support several modes of operation. For example, a streaming video player will probably support several frame rates and window sizes. Larger window sizes and higher frame rates require larger guarantees, and will be assigned lower importance. This is similar to the resource management scheme described by Oparah [15].

Let the *total importance* of a configuration be the sum of the importances of the running applications. A *sub-configuration* of a given configuration has the same set of applications, but may give them weaker guarantees. When asked to switch to an infeasible configuration, the RM must search for a "good" feasible configuration to switch to instead. Good configurations are sub-configurations of the requested configuration that have high total importance, that are not too different from the current configuration, and that satisfy the current *configuration policies*. Configuration policies are user-specified policies about configurations. For example, the system administrator might specify that a user may be guaranteed no more than 20% of the processor time, that the default time-sharing scheduler must be allowed the opportunity to run at least every 100ms, and that only threads belonging to a certain activity may be scheduled by a given sub-scheduler of the Rialto/NT scheduler.

5 Proposed Work

I will first implement a restricted version of the loadable scheduler infrastructure in Windows NT, along with a few schedulers. This prototype will support hierarchical schedulers, but will not implement the full resource management infrastructure.

Next, I plan to implement the entire infrastructure in Linux, FreeBSD, or Windows NT, including the full resource manager. Note that to show scheduler portability, it is necessary to implement hierarchical scheduling, but not the entire resource manager, in multiple operating systems.

In parallel with the implementations, I want to find some applications that require application-specific scheduling that I can use to evaluate the performance of loadable schedulers. For example, an mpeg player, a parallel program for a symmetric multiprocessor, and a web server that creates a number of processes.

5.1 Non-issue: Security

A misbehaving scheduler could cause problems in several ways: by reading from out-of-bounds memory it could cause a security violation, by writing to out-of-bounds memory or overflowing a kernel stack it could crash the OS, and by failing to respond to a query or by failing to release spinlocks it could lock up the system. Although the OS community has addressed the issue of surviving misbehaved kernel extensions in great detail, this work is

unfortunately not available in any mainstream OS, and replicating it is beyond the scope of my work.

I take the practical approach that is already taken by Linux, Solaris, and Windows NT with respect to loadable kernel modules: trust the module author. The act of loading a scheduler may be triggered by an untrusted user, but the scheduler will have been cryptographically signed or placed in a secure location by someone with administrative privileges.

Schedulers can also fail functionally, by not meeting the guarantees that they have given. This will adversely affect threads and schedulers under them in the hierarchy, but will not affect sibling schedulers or their children. In the example, the Rialto/NT scheduler could starve thread T1, but because it cannot use more CPU time than it is given by the fixed-quantum scheduler, it cannot steal time from the fair-share scheduler.

5.2 Non-issue: Resources Other than CPU

Of course, in addition to intelligently scheduling the CPU, we would also like to schedule memory, disk and network bandwidth, I/O bus bandwidth, and other resources that applications require in order to execute predictably. I believe that the loadable scheduler architecture can be extended to schedule other resources, but doing this is beyond the scope of my work.

5.3 Non-issue: Fixing the OS

Even when applications are not blocked for want of non-CPU resources, there are CPU-related issues that can prevent predictable scheduling of user code. For example, non-scheduled use of processor time by interrupt handlers and other kernel routines, as well as non-preemptible sections of kernel code can both contribute to scheduling latency—the time between a scheduling decision and when the thread actually begins to run. In operating systems that were not designed for real-time, scheduling latency can sometime be high; see [12] for a study of scheduling latency in Windows NT.

I assume that the operating systems in question have low enough scheduling latency that useful soft real-time scheduling can be done. Fixing latency and overhead problems is not part of the proposed work—high latencies will simply limit the timing granularity that applications can use. However, schedulers should be coded defensively in order to cope gracefully with timing jitter in the system.

6 Research Questions

These are issues that I will address as part of the dissertation work.

1. What view of time should a scheduler have? Of course, RT schedulers need to know about real time. Other schedulers may only want to keep track of time that accumulates when they are actually in control of a processor. Are there schedulers for which neither of these is appropriate?
2. Closely related to the previous question: what changes are necessary to make existing scheduling algorithms work as non-root schedulers? That is, with less than 100% of the CPU? This should not be a problem for real-time schedulers with a map of future time—they can simply take the “holes” into account explicitly. Can all non-RT schedulers just ignore the holes?
3. Part of the proposed work is to try to make schedulers robust across changes of assumption between different operating systems. It may be that some kinds of schedulers are very difficult to write portably—I will consider it acceptable to have some negative results here. That is, schedulers that cannot be easily written in a portable way.
4. How should priority inversion be avoided? CPU inheritance scheduling [7] uses a generalized form of priority inheritance to avoid the problem—I may be able to use a similar approach. However, this is probably very difficult to graft onto all of the synchronization mechanisms in an existing operating system.
5. I currently plan is to have one scheduler hierarchy for all processors. So, the system supports time-sharing but not space-sharing. If they want to, schedulers can chose to schedule only a subset of the available processors. Is explicit support for space-sharing desirable?
6. Can a tool extract guarantees from scheduler code automatically, or verify (on-line or off-line) that schedulers can meet the guarantees that they make?

7. What is the language in which configuration policies are expressed? Possible options are a formal specification language, and fragments of general-purpose code.
8. What should be the role of reflective information in the loadable scheduler scheme? That is, how much information about the scheduling hierarchy itself should be available to applications and schedulers?

7 Evaluation

My hypotheses are:

1. The hierarchical loadable scheduler architecture reduces the cost of implementing new scheduling policies in general-purpose OS kernels, and results in a more flexible system than one based on custom scheduler work.
2. The architecture enables efficient execution of applications requiring specialized schedulers.
3. The resource manager makes it possible for the system to dynamically respond to changing conditions, and to provide high perceived value to the user or users.

The "reduced cost" part of the first hypothesis is a software-engineering argument, and is difficult to test. I believe that it can be settled by anecdotal evidence and by presenting examples of loadable scheduler source code. That loadable schedulers are more flexible than one-shot special purpose schedulers follows from the design of the system.

To test the second hypothesis I will develop application-specific metrics for determining efficiency. For example, number of missed deadlines for a streaming video player, response time to user input of an interactive program, and speedup of a parallel program. I will run the applications under the default time-sharing scheduler in an unmodified OS and under a suitable loadable scheduler, and compare the results.

I will also run microbenchmarks to measure the speed of operations such as the time to schedule a thread—this is an important performance metric because it is in the critical path for system services. It is less crucial for other parts of the scheduling infrastructure such as the resource manager to be fast, since they will run much less often. Loadable schedulers should not introduce significant overhead for legacy applications that do not make use of loadable schedulers at all.

Evaluating the resource manager is more difficult. That it provides high perceived value is testable, if we assume that users have correctly assigned importances to applications. The policies given to the RM should be expressive enough to implement useful policies. The RM itself should be unobtrusive in that it does not overly complicate the programming model or use undue amounts of CPU time while searching for good system configurations.

8 Conclusion

Lack of domain-specific knowledge prevents time-sharing schedulers from efficiently scheduling some classes of applications when there is contention for the CPU. By integrating support for flexible application-specific scheduling into common operating systems, I expect to increase the range of applications that can be run on general-purpose machines, thereby increasing the perceived value of the system to users.

References

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[2] Gaurav Banga, Peter Druschel, and Jeffery C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[4] Intel Corporation. Soft Migration. `http://developer.intel.com/ial/sm/`.

[5] Zhong Deng, Jane W.-S. Liu, Lynn Zhang, Seri Mouna, and Alban Frei. An Open Environment for Real-Time Applications. *Real-Time Systems Journal*, 16(2/3):165–185, May 1999.

[6] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[7] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 91–105, Seattle, WA, October 1996.

[8] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design*. Prentice-Hall, 1994.

[9] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pages 107–121, Seattle, WA, October 1996.

[10] Michael B. Jones, Paul J. Leach, Richard P. Draves, and Joseph S. Barrera, III. Modular Real-Time Resource Management in the Rialto Operating System. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems*, May 1995.

[11] Michael B. Jones and John Regehr. Issues in Using Commodity Operating Systems for Time-Dependent Tasks: Experiences from a Study of Windows NT. In *Proc. of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[12] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proc. of the 7th Workshop on Hot Topics in Operating Systems*, pages 96–101, March 1999.

[13] Hiroyuki Kaneko, John A. Stankovic, Subhabrata Sen, and Krithi Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[14] Jason Nieh, James G. Hanko, J. Duane Northcutt, and Gerard A. Wall. SVR4 UNIX Scheduler Unacceptable for Multimedia Applications. In *Proc. of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video*, November 1993.

[15] Don Oparah. Adaptive Resource Management in a Multimedia Operating System. In *Proc. of the 8th International Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[16] SCO. Uniform Driver Interface White Paper Rev 0.85, November 1998.

[17] John A. Stankovic. The Many Faces of Multi-Level Real-Time Scheduling. In *Proc. of the 2nd International Workshop on Real-Time Computing Systems and Applications*, Tokyo, October 1995.

[18] 4Front Technologies. SoftOSS. `http://www.opensound.com/softoss.html`.